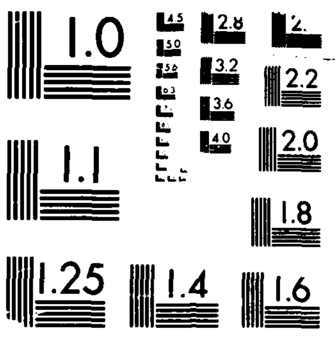
AD-A193 715 ADA (TRADE NAME) COMPILER VALIDATION SUMMARY REPORT: ALSYS LTD ALSYCOMP -.. (U) NATIONAL COMPUTING CENTRE LTD MANCHESTER (ENGLAND) 10 JUN 87 1/1

UNCLASSIFIED F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART

URFAU [illegible] STANDARDS 1963 A

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

DTIC FILE COPY

(2)

AD-A193 715

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETEING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Ada Compiler Validation Summary Report: ALSYS Ltd AlsyCOMP_007, VI.0 Host: HP 1000 A900 Target: Same as Host		5. TYPE OF REPORT & PERIOD COVERED 18 June'87 to 18 June'88
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) National Computing Centre Ltd; M1 7ED UK		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS National Computing Centre Ltd; Oxford Rd., Manchester M1 7Ed UK		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081ASD/SIOL		12. REPORT DATE 18 June 1987
		13. NUMBER OF PAGES 48p.
14. MONITORING AGENCY NAME & ADDRESS *(If different from Controlling Office)* National Computing Centre Limited M1 7ED UK		15. SECURITY CLASS *(of this report)* UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20. If different from Report)*

UNCLASSIFIED

DTIC SELECTED APR 04 1988 S E D

18. SUPPLEMENTARY NOTES

19. KEYWORDS *(Continue on reverse side if necessary and identify by block number)*

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

See Attached.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the AlsyCOMP_007, V1.0 using Version 1.8 of the Ada* Compiler Validation Capability (ACVC). The AlsyCOMP_007, is hosted on a HP 1000 A900 operating under RTE-A, 5.0. Programs processed by this compiler may be executed on a HP 1000 A900 operating under RTE-A 5.0.

On-site testing was performed 15 June 1987 through 18 June 1987 at ALSYS Ltd, Henley-on-Thames, under the direction of The National Computing Centre (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2138 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 242 executable tests that make use of floating-point precision exceeding that supported by the implementation were not processed. After the 2138 tests were processed, results for Class A, C, D, or E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 33 of the processed tests determined to be inapplicable; The remaining 2105 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	96	222	298	241	161	97	134	261	121	32	217	225	2105
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	20	103	122	6	0	0	5	1	9	0	1	8	275
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

AVF Control Number: AVF-VSR-90502/22

Ada* COMPILER
VALIDATION SUMMARY REPORT
ALSYS Ltd
AlsyCOMP_007, V1.0
Host: HP 1000 A900
Target: Same as Host

Completion of On-Site Testing
18 June 1987

Prepared By:
The National Computing Centre Limited
Oxiord Road
Manchester
M1 7ED
UK

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C.
USA

Accession For		
NTIS GRA&I		X
DTIC TAB		☐
Unannounced		☐
Justification		
By		
Distribution/		
Availability Codes		
Dist	Avail and/or Special	
A-1		

QUALITY INSPECTED 4

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

88 4 4 059

Ada* Compiler Validation Summary Report:

Compiler Name: AlsyCOMP_007

Host:	Target:
HP 1000 A900 under RTE-A 5.0	Same as Host

Testing Completed 18 June 1987 using ACVC 1.8

This report has been reviewed and is approved.

[illegible]

The National Computing Centre Ltd
Vony Gwillim
Oxford Road
Manchester
M1 7ED

[illegible]

Ada Validation Office
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA

Virginia L. Castor

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the AlsyCOMP_007, V1.0 using Version 1.8 of the Ada* Compiler Validation Capability (ACVC). The AlsyCOMP_007, is hosted on a HP 1000 A900 operating under RTE-A, 5.0. Programs processed by this compiler may be executed on a HP 1000 A900 operating under RTE-A 5.0.

On-site testing was performed 15 June 1987 through 18 June 1987 at ALSYS Ltd, Henley-on-Thames, under the direction of The National Computing Centre (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2138 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 242 executable tests that make use of floating-point precision exceeding that supported by the implementation were not processed. After the 2138 tests were processed, results for Class A, C, D, or E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 33 of the processed tests determined to be inapplicable; The remaining 2105 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	96	222	298	241	161	97	134	261	121	32	217	225	2105
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	20	103	122	6	0	0	5	1	9	0	1	8	275
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from characteristics of particular operating systems, hardware, or implementation strategies. All of the dependencies demonstrated during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behaviour that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any unsupported language constructs required by the Ada Standard.
- To determine that the implementation-dependent behaviour is allowed by the Ada Standard

Testing of this compiler was conducted by NCC under the direction of the AVF according to policies and procedures established by the Ada Validation Organisation (AVO). On-site testing was conducted from 15 June 1987 through 18 June 1987 at ALSYS Ltd, Henley-on- Thames.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. 552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organisations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
The National Computing Centre Ltd
Oxford Road
Manchester
M1 7ED
United Kingdom

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, FEB 1983.

2. Ada Validation Organization: Policies and Procedures, MITRE Corporation, JUN 1982, PB 83-110601.

3. Ada Compiler Validation Capability Implementer's Guide, SofTech, Inc., DEC 1984.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. A set of programs that evaluates the conformity of a compiler to the Ada language specification, ANSI/MIL-STD-1815A.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	The National Computing Centre Ltd. In the context of this report, the AVF is responsible for conducting compiler validations according to established policies and procedures.
AVO	The Ada Validation Organization. In the context of this report, the AVO is responsible for setting procedures for compiler validations.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	A test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.

Host	The computer on which the compiler resides.
Inapplicable test	A test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	A test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn	A test found to be incorrect and not used to check conformity to test the Ada language specification. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. However, no checks are performed during execution to see if the test objective has been met. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capabilities of a compiler. Since there are no requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimization allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of these units is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation are listed in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: AlsyCOMP_007, V1.0

ACVC Version: 1.8

Certification ~~Expiration Date: To be inserted~~ #: 870615N1.08074

Host Computer:

Machine :	HP 1000 A900
Operating System:	RTE-A 5.0
Memory Size:	6M/9M *

Target Computer:

Same as Host.

Communications Network: Not applicable.

* Two computers were used, having 6M bytes and 9 M bytes of memory.

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behaviour of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. This compiler is characterized by the following interpretations of the Ada Standard:

- Capacities.

  The compiler correctly processes compilations containing loop statements nested to 33 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation does not reject such calculations and processes them correctly. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

  This implementation supports the additional predefined types LONG_INTEGER, and LONG_FLOAT, in the package STANDARD. (See tests B86001C and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Array Types.

  An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/ or SYSTEM.MAX_INT.

  A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises STORAGE_ERROR when the array type is declared. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises STORAGE_ERROR when the array type is declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation accepts the declaration. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications during compilation. (See test E38104A.)

In assigning record types with discriminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are not evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

. Functions

An implementation may allow the declaration of a parameterless function and an enumeration literal having the same profile in the same immediate scope, or it may reject the function declaration. If it accepts the function declarations, the use of the enumeration literal's identifier denotes the function. This implementation rejects the declarations. (See test E66001D.)

. Representation clauses.

The Ada Standard does not require an implementation to support representation clauses. If a representation clause is not supported, then the implementation must reject it. While the operation of representation clauses is not checked by Version 1.8 of the ACVC, they are used in testing other language features. This implementation rejects 'SIZE and 'STORAGE_SIZE for tasks, 'STORAGE_SIZE for collections, and 'SMALL clauses. Enumeration representation clauses appear not to be supported. (See tests C55B16A, C87B62A, C87B62B, C87B62C, and BC1002A.)

. Pragmas.

The pragma INLINE is not supported for procedures. The pragma INLINE is not supported for functions. (See tests CA3004E and CA3004F.)

. Input/Output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants. The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, AE2101H, CE2201D, CE2201E, and CE2401D.)

An existing text file can be opened in OUT_FILE mode, cannot be created in OUT_FILE mode, and cannot be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for reading only. (See tests CE3111A.E (5 tests).)

More than one internal file can be associated with each external file for sequential I/O for both reading and writing. (See tests CE2107A..F (6 tests).)

More than one internal file can be associated with each external file for direct I/O for both reading and writing. (See tests CE2107A..F (6 tests).)

An external file associated with more than one internal file cannot be deleted. (See test CE2110B.)

Temporary sequential files are given a name. Temporary direct files are given a name. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

. Generics.

Generic subprogram bodies can only be compiled in separate compilations provided that no instantiation of the corresponding generic occur prior to the compilation of the generic body. (See test CA2009F.)

Generic package bodies can only be compiled in separate compilations provided that no instantiation of the corresponding generic occur prior to the compilation of the generic body. (See tests CA2009C and BC3205D.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.8 of the ACVC contains 2399 tests. When validation testing of AlsyCOMP_007 was performed, 19 tests had been withdrawn. The remaining 2380 tests were potentially applicable to this validation. The AVF determined that 275 tests were inapplicable to this implementation, and that the 2105 applicable tests were passed by the implementation.

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	68	862	1105	15	11	44	2105
Failed	0	0	0	0	0	0	0
Inapplicable	1	5	263	2	2	2	275
Withdrawn	0	7	12	0	0	0	19
TOTAL	69	874	1380	17	13	46	2399

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER												
	2	3	4	5	6	7	8	9	10	11	12	14	TOTAL
Passed	96	222	298	241	161	97	134	261	121	32	217	225	2105
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	20	103	122	6	0	0	5	1	9	0	1	8	275
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

3.4 WITHDRAWN TESTS

The following 19 tests were withdrawn from ACVC Version 1.8 at the time of this validation:

C32114A	C41404A	B74101B
B33203C	B45116A	C87B50A
C34018A	C48008A	C92005A
C35904A	B49006A	C940ACA
B37401A	B4A010C	CA3005A..D (4 tests)
		BC3204C

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. For this validation attempt, 275 tests were inapplicable for the reasons indicated:

- C34001D, B52004E, B55B09D, and C55B07B use SHORT_INTEGER which is not supported by this implementation.

- C34001F, and C35702A use SHORT_FLOAT which is not supported by this implementation.

- C55B16A makes use of an enumeration representation clause containing noncontiguous values which is not supported by this implementation.

. D55A03G..H (2 tests) contain loops nested to depths greater than 33. This exceeds the capacity of the implementation.

. B86001DT requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

. C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

. C87B62A..C (3 tests) use length clauses which are not supported by this implementation.

. C92005B assumes that the default storage size for a task belongs to type INTEGER. This is not the case for this implementation.

. BA2001E requires that duplicate names of subunits with a common ancestor be detected at compilation time. This compiler correctly detects the error at link time, and the AVO rules that such behaviour is acceptable.

. CA2009C, CA2009F, and BC3205D compile generic subunits in separate compilation files. Separate compilation of generic specifications and bodies is only supported by this compiler if no instantiation of the corresponding generic occurs prior to the compilation of the generic body.

. CA3004E, EA3004C, and LA3004A use pragma INLINE for procedures which is not supported by this implementation.

. CA3004F, EA3004D, and LA3004B use pragma INLINE for functions which is not supported by this implementation.

. AE2101H, CE2401D use instantiation of package DIRECT_IO with unconstrained array types which is not supported by this implementation.

. CE2107E determines whether sequential and direct files may be associated with the same external file. This is not the case for this implementation.

. CE3111B..E (4 tests) are inapplicable because multiple internal files cannot be associated with the same external file, when one file is open for writing. The proper exception is raised when multiple access is attempted.

. CE3114B is inapplicable because multiple internal files cannot be associated with the same external file for delete.

. The following 242 tests make use of floating-point precision that exceeds the maximum of 9 supported by the implementation:

C24113F..Y (20 tests)	C45241F..Y (20 tests)
C35705F..Y (20 tests)	C45321F..Y (20 tests)
C35706F..Y (20 tests)	C45421F..Y (20 tests)
C35707F..Y (20 tests)	C45424F..Y (20 tests)
C35708F..Y (20 tests)	C45521F..Z (21 tests)
C35802F..Y (20 tests)	C45621F..Z (21 tests)

3.6 SPLIT TESTS

If one or more errors do not appear to have been detected in a Class B test because of compiler error recovery, then the test is split into a set of smaller tests that contain the undetected errors. These splits are then compiled and examined. The splitting process continues until all errors are detected by the compiler or until there is exactly one error per split. Any Class A, Class C, or Class E test that cannot be compiled and executed because of its size is split into a set of smaller subsets that can be processed.

Splits were required for 14 Class B tests.

B32202A	B43201D	B62001D
B32202B	B45102A	B91004A
B32202C	B61012A	B95069A
B33001A	B62001B	B95069B
B37004A	B62001C	

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.8 produced by AlsyCOMP_007, was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behaviour on all inapplicable tests.

3.7.2 Test Method

Testing of AlsyCOMP_007 using ACVC Version 1.8 was conducted on-site by a validation team from the AVF. The configuration consisted of two HP 1000 A900s operating under RTE-A V5.0.

A magnetic tape containing all tests was taken on-site by the validation team for processing. The magnetic tape contained tests that make use of implementation-specific values which were customized before being written to the magnetic tape. Tests requiring splits during the prevalidation testing were not included in their split form on the magnetic tape. The contents of the magnetic tape were loaded first onto a VAX 750 computer, where the required splits were performed using the VAX editor (EDT) using prepared command scripts. The processed source files were then transferred to an *HP 9000* via an Ethernet connection, where two cartridge tapes were produced of the processed source files. Each cartridge tape was then loaded onto a HP 1000 A900.

After the test files were loaded to disk, the full set of tests was compiled and linked and all executable tests were run on the two HP 1000 A900s. Results were transferred via an Ethernet connection to a VAX 750 computer for printing.

The compiler was tested using command scripts provided by ALSYS Ltd and reviewed by the validation team.

Tests were compiled, linked and executed (as appropriate) using two computers, which were both host and target. Test output, compilation listings, and job logs were captured on magnetic tape and archived at AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

The validation team arrived at ALSYS Ltd, Henley-on-Thames on 15 June 1987 and departed after testing was completed on 18 June 1987.

APPENDIX A

COMPLIANCE STATEMENT

ALSYS Ltd has submitted the following compliance statement concerning the AlsyCOMP_007.

Compliance Statement

Base Configuration:

Compiler: AlsyCOMP_007, V1.0

Test Suite: Ada* Compiler Validation Capability, Version 1.8

Host Computer:

Machine:	HP 1000 A900
Operating System:	RTE-A 5.0

Target Computer:

Machine:	Same as Host

Communications Network:	Not applicable

ALSYS Ltd. has made no deliberate extensions to the Ada language standard.

ALSYS Ltd. agrees to the public disclosure of this report.

ALSYS Ltd. agrees to comply with the Ada trademark policy, as defined by the Ada Joint Program Office.

[illegible] Date: 16/6/87

ALSYS Ltd
M L J Jordan
Marketing Director

*Ada is registered trademark of the United States Government (Ada Joint Program Office).

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of MIL-STD-1815A, and to certain allowed restrictions on representation classes. The implementation-dependent characteristics of the AlsyCOMP_007, V1.0 are described in the following sections which discuss topics one through eight as stated in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A). The implementation-specific portions of the package STANDARD are also included in this appendix.

```
Package STANDARD is

   ...

   type LONG_INTEGER is -2_147_483_648   .. 2_147_483_647;
   type INTEGER is range -32_768 .. 32_767;

   type FLOAT is digits 6 range
        -(1.0-2.0**-23)*2.0**124 .. (1.0-2.0**-23)*2.0**124;
   type LONG_FLOAT is digits 9 range
        -(1.0-2.0**-55)*2.0**124 .. (1.0-2.0**-55)*2.0**124;

   type DURATION is delta 2.0**-14 range -86_400.0..86_400.0;

end STANDARD;
```

Alsys HP 1000 Ada* Compiler

Appendix F
Implementation - Dependent Characteristics

Version 1.0

Alsys S.A.
29, Avenue de Versailles
78170 La Celle St. Cloud, France

Alsys, Inc.
1432 Main Street
Waltham, MA 02154, U.S.A.

Alsys Ltd.
Partridge House, Newtown Road
Henley-on-Thames,
Oxfordshire RG9 1EN, U.K.

Printed: May 1987

PREFACE

This *Alsys HP 1000 Ada Compiler Appendix F* is for programmers, software engineers, project managers, educators and students who want to develop an Ada program for any Hewlett-Packard 1000 processor that runs RTE-A.

This appendix is a required part of the *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815A, February 1983 (throughout this appendix, citations in square brackets refer to this manual). It assumes that the user is already familiar with the RTE-A operating system, and has access to the following HP documents:

RTE-A User's Manual, 92077-90002

TABLE OF CONTENTS

Appendix F

Implementation-Dependent Characteristics

This appendix summarises the implementation-dependent characteristics of the Alsys HP 1000 Ada Compiler.

The sections of this appendix are as follows:

1. The form, allowed places, and effect of every implementation-dependent pragma.
2. Restrictions on predefined pragmas.
3. The name and type of every implementation-dependent attribute.
4. Restrictions on predefined attributes.
5. The specification of the package SYSTEM.
6. The list of all restrictions on representation clauses.
7. The conventions used for any implementation-generated names denoting implementation-dependent components.
8. The interpretation of expressions that appear in address clauses, including those for interrupts.
9. Any restrictions on unchecked conversions.
10. Any implementation-dependent characteristics of the input-output packages.
11. Characteristics of numeric types.
12. Layout of data types in memory.
13. Other implementation-dependent characteristics.
14. Compiler limitations.

The name *Ada Run-Time Executive* refers to the run-time library routines provided for all Ada programs. These routines implement the Ada heap, exceptions, tasking control, and other utility functions.

1 Implementation-Dependent Pragmas

Ada programs can interface with subprograms written in various languages, including FORTRAN, Pascal and Macro/1000 through the use of the predefined pragma INTERFACE [13.9] and the implementation-defined pragma INTERFACE_NAME.

1.1 INTERFACE

Pragma INTERFACE specifies the name of an interfaced subprogram and the name of the programming language for which calling and parameter passing conventions will be generated. Pragma INTERFACE takes the form specified in the *Language Reference Manual*:

pragma INTERFACE (*language_name*, *subprogram_name*);

where

- *language_name* is the name of the other language whose calling and parameter passing conventions are to be used.
- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram

The language names currently accepted by pragma INTERFACE are FTN7X, PAC, MACRO, DIRECT, DIRECT_SKIP_NUMERIC_ERROR and SKIP_PROGRAM_ERROR.

Parameter and Result Passing Conventions

The following Ada types are not supported when passing parameters to, nor returning results from, interfaced subprograms:

- task types.
- private types.
- fixed-point types

The result type of an interfaced function is further restricted to be a type of fixed size: unconstrained arrays or records are not allowed.

Three parameter-passing modes are supported for interfaced subprograms:

- **in**
- **in out**
- **out**

When passing scalar types to formal parameters of mode **in** or **in out** each scalar type is passed as a copy of its value. If a parameter of scalar type is passed with mode **in out** or **out**, then code planted after the call updates the parameter on return from the subprogram. Access types are handled similarly except that they can only be passed with mode **in**.

Array and record (composite) types are passed by reference, regardless of the parameter passing mode. The Ada compiler expects the object to be in logical address space; if not (that is, the object is in VMA) then the parameter type should be declared as SYSTEM.ADDRESS and the ADDRESS attribute should be used for the corresponding parameter in the call.

An object will be in logical address space if it meets one of the following sets of criteria:

- The object is local to the current subprogram and its size is not greater than the "cut-off" size for local objects (see option LOCAL_MAX in the compile command).
- The object is global and its size is not greater than the "cut-off" size for global objects (see option GLOBAL_MAX in the compile command).
- The object is "uplevel" (that is, declared in an enclosing subprogram), its size is not greater than the "cut-off" size for local objects, and a no-tasking predefined library is being used.

Note that no consistency checking is performed between the subprogram parameters as declared in Ada and the corresponding parameters in the interfaced subprogram.

If the 'ADDRESS of any object is passed to an interfaced subprogram, that object is not protected from being updated by the interfaced subprogram. Such objects, as well as parameters passed with mode **in out** or **out** (and **in** for arrays and records), will have no run-time checks performed on their contents upon return from the interfaced subprogram. Erroneous program behaviour may result if the parameter values are altered such that they violate Ada constraints for the type of the actual parameter.

Result Returned by Interfaced Function Call

Only objects of a fixed size can be returned as the result of an interfaced function call. In other words, unconstrained arrays and records cannot be returned as the result of a call. All scalar objects except LONG_FLOAT are returned in registers. LONG_FLOAT return results are processed using a temporary created by the caller.

Interface Language Names

The following table defines the characteristics of the accepted interface language names.

Name	PCAL type	Strings	Comments	Recommended Use
FTN7X	default PCAL	FTN7X strings	Packed strings with descriptor	For a Fortran subprogram
PAC	default PCAL	packed		For any subprogram requiring packed strings without a Fortran string descriptor
MACRO	default PCAL	unpacked		For any subprogram requiring unpacked strings

DIRECT	PCALN	packed		Non-CDS calls
DIRECT_ SKIP_ NUMERIC_ ERROR	PCALN	packed	Same as DIRECT except for an error return which raises predefined exception NUMERIC_ERROR	Mathematical functions
SKIP_ PROGRAM_ ERROR	default PCAL	packed	Same as PAC except for an error return which raises predefined exception PROGRAM_ERROR	System calls with Abort

1.2 INTERFACE_NAME

Pragma INTERFACE_NAME associates the name of an interfaced subprogram, as declared in Ada, with its name in the language of origin. If pragma INTERFACE_NAME is not used, then the name of the subprogram specified in the pragma INTERFACE is used, with all alphabetic characters shifted to lower case. This pragma takes the form

pragma INTERFACE_NAME (*subprogram_name*, *string_literal*);

where

- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.
- *string_literal* is the name by which the interfaced subprogram is referred to at link-time. This name should be given with alphabetic characters in upper case for compatibility with names generated by other tools.

The use of INTERFACE_NAME is optional, and is not needed if a subprogram has the same name in Ada as in the language of origin. It is useful, for example, if the name of the subprogram in its original language contains an upper case letter, or characters that are not permitted in Ada identifiers. Ada identifiers can contain only letters, digits and underscores, whereas the RTE-A linker allows external names to contain other characters, e.g. the period. These characters can be specified in the *string_literal* argument of the pragma INTERFACE_NAME. Note also that, by default, Macro/1000 shifts identifiers to upper case, so this should be reflected in the string literal.

The pragma INTERFACE_NAME is allowed at the same places of an Ada program as the pragma INTERFACE [13.9]. However, the pragma INTERFACE_NAME must always occur after the pragma INTERFACE declaration for the interfaced subprogram.

To avoid conflicts with the Ada Run-Time Executive, the names of interfaced subprograms should not begin with the letters "ADA".

Example

```
package SAMPLE_LIB is
        function SAMPLE_DEVICE (X : INTEGER) return INTEGER;
        function PROCESS_SAMPLE (X : INTEGER) return INTEGER;
private
        pragma INTERFACE (FTN7X, SAMPLE_DEVICE);
        pragma INTERFACE (PAC, PROCESS_SAMPLE);
        pragma INTERFACE_NAME (SAMPLE_DEVICE, "DEV10");
        pragma INTERFACE_NAME (PROCESS_SAMPLE, "PSAMPLE");
end SAMPLE_LIB;
```

1.3 Other Pragmas

No other implementation-dependent pragmas are supported in the current version of this compiler.

2 Predefined Language Pragmas

The predefined language pragmas are implemented as defined in the *Language Reference Manual* [B], with the following exceptions:

Any use of the predefined language pragmas

CONTROLLED
INLINE
MEMORY_SIZE
OPTIMISE
PACK
SHARED
STORAGE_UNIT
SYSTEM_NAME

is either ignored without comment (the pragma has no effect) or elicits a warning from the compiler:

```
This pragma is not currently supported by the
implementation.
```

The pragma SUPPRESS applies to the whole of a compilation; it cannot be used to omit checks within parts of the source code.

3 Implementation-Dependent Attributes

There are no implementation-dependent attributes available to the user.

4 Predefined Language Attributes

The predefined language pragmas are implemented as defined in the *Language Reference Manual* [A], with the following exception:

P'STORAGE_SIZE

When applied to an access type or subtype of an access type the compiler produces one of two responses:

- If there is no length clause, the compiler issues the warning:

  ```
  The prefix of the attribute STORAGE_SIZE is an
  access type which has no length clause.  The
  value returned by STORAGE_SIZE is always 0.
  ```

- If a length clause is specified, the compiler issues the error:

  ```
  Length clauses are not currently implemented.
  ```

5 Specification of the Package SYSTEM

```
package SYSTEM is

    type ADDRESS is new LONG_INTEGER;

    type NAME is (HP1000_RTE_A);
    SYSTEM_NAME       : constant NAME := NAME'FIRST;

    STORAGE_UNIT      : constant := 16;
    MEMORY_SIZE       : constant := 2**26; -- 128Mbytes for VMA
    MIN_INT           : constant := -(2**31);
    MAX_INT           : constant :=  2**31-1;
    MAX_DIGITS        : constant := 9;
    MAX_MANTISSA      : constant := 31;
    FINE_DELTA        : constant := 2#1.0#E-31;
    TICK              : constant := 0.01;

    subtype PRIORITY is INTEGER range 0 .. 10;

    -- Implementation-defined declarations:

    -- There are three kinds of address in this implementation:
    --    (a) VMA data address
    --    (b) Logical data address
    --    (c) Code address.
    -- The 'ADDRESS attribute in Ada yields an object of type ADDRESS (which is defined
    -- above) for all three categories:

    -- For category (a), this is the full VMA address of the object.
```

```
-- For category (b), the logical address is contained in the high-addressed word. The low
   -addressed word has the local-reference bit (bit 0) set:

type LOGICAL_ADDRESS is new NATURAL;
type ADDRESS_IN_LOGICAL is
  record
    DUMMY      : INTEGER,  -- Local reference bit set
    LOG_ADDR   : LOGICAL_ADDRESS;
  end record;

-- For category (c), the low addressed word contains the segment number and STT index
-- fields (suitable for PCALV) and the high-addressed word contains the 15-bit offset within
-- the segment

type CODE_LABEL is new INTEGER;  -- 16-bit Seg no / STT index
type CODE_ADDRESS is
  record
    LABEL   : CODE_LABEL;
    OFFSET  : NATURAL;
  end record;

-- If the 'ADDRESS attribute is applied to a data object known to be in logical data space, or
-- to a code address, then the user can perform an UNCHECKED_CONVERSION from type
-- ADDRESS to the appropriate record type above to obtain the components.

NULL_ADDRESS : constant ADDRESS := 0;

-- Subprograms to read from and write to both logical data space and VMA for INTEGER
-- and LONG_INTEGER:

function READ_LOGICAL_ADDRESS_SINGLE (X : in LOGICAL_ADDRESS)
                                               return INTEGER;
       pragma INTERFACE (ADA, READ_LOGICAL_ADDRESS_SINGLE);
       pragma INTERFACE_NAME (READ_LOGICAL_ADDRESS_SINGLE,
                                     "ADA.READLOGSINGL");

function READ_LOGICAL_ADDRESS_DOUBLE (X : in LOGICAL_ADDRESS)
                                               return LONG_INTEGER;
       pragma INTERFACE (ADA, READ_LOGICAL_ADDRESS_DOUBLE);
       pragma INTERFACE_NAME (READ_LOGICAL_ADDRESS_DOUBLE,
                                     "ADA.READLOGDOUBL");

function READ_VMA_ADDRESS_SINGLE (X : in ADDRESS) return INTEGER;
       pragma INTERFACE (ADA, READ_VMA_ADDRESS_SINGLE);
       pragma INTERFACE_NAME (READ_VMA_ADDRESS_SINGLE,
                                     "ADA.READVMASINGL");

function READ_VMA_ADDRESS_DOUBLE (X : in ADDRESS)
                                               return LONG_INTEGER;
       pragma INTERFACE (ADA, READ_VMA_ADDRESS_DOUBLE);
       pragma INTERFACE_NAME (READ_VMA_ADDRESS_DOUBLE,
                                     "ADA.READVMADOUBL");
```

```
procedure WRITE_LOGICAL_ADDRESS_SINGLE (X : in LOGICAL_ADDRESS;
                                         Y : in INTEGER);
    pragma INTERFACE (ADA, WRITE_LOGICAL_ADDRESS_SINGLE);
    pragma INTERFACE_NAME (WRITE_LOGICAL_ADDRESS_SINGLE,
                           "ADA.WRITELOGSNGL");

procedure WRITE_LOGICAL_ADDRESS_DOUBLE (X : in LOGICAL_ADDRESS;
                                         Y : in LONG_INTEGER);
    pragma INTERFACE (ADA, WRITE_LOGICAL_ADDRESS_DOUBLE);
    pragma INTERFACE_NAME (WRITE_LOGICAL_ADDRESS_DOUBLE,
                           "ADA.WRITELOGDOUB");

procedure WRITE_VMA_ADDRESS_SINGLE (X : in ADDRESS;
                                     Y : in INTEGER);
    pragma INTERFACE (ADA, WRITE_VMA_ADDRESS_SINGLE);
    pragma INTERFACE_NAME (WRITE_VMA_ADDRESS_SINGLE,
                           "ADA.WRITEVMASNGL");

procedure WRITE_VMA_ADDRESS_DOUBLE (X : in ADDRESS;
                                     Y : in LONG_INTEGER);
    pragma INTERFACE (ADA, WRITE_VMA_ADDRESS_DOUBLE);
    pragma INTERFACE_NAME (WRITE_VMA_ADDRESS_DOUBLE,
                           "ADA.WRITEVMADOUB");

end SYSTEM;
```

6 Restrictions on Representation Clauses

Representation clauses [13.1] are not supported by this version of the Alsys HP 1000 Ada Compiler. Any program containing a representation clause is rejected at compilation time. The pragma PACK [13.1] is also not supported. However, its presence in a program does not in itself make the program illegal; the Compiler will simply issue a warning message and ignore the pragma.

7 Conventions for Implementation-Generated Names

There are no implementation-generated names [13.4] available to the user in the current version of the Alsys HP 1000 Ada Compiler.

The following predefined library units cannot be recompiled:

ALSYS_ADA_RUNTIME
ALSYS_BASIC_IO
ALSYS_BINARY_IO
ALSYS_COMMON_IO
ALSYS_FILE_MANAGEMENT
ALSYS_SYS_IO
CALENDAR
DIRECT_IO

IO_EXCEPTIONS
SEQUENTIAL_IO
STANDARD
SYSTEM
TEXT_IO
UNCHECKED_CONVERSION
UNCHECKED_DEALLOCATION

8 Address Clauses

Address clauses [13.5] are not supported in this version of the Alsys HP 1000 Ada Compiler.

9 Restrictions on Unchecked Conversions

Unchecked conversions [13.10.2] are allowed only between types which have the same value for their 'SIZE attribute.

10 Input-Output Packages

The predefined input-output packages SEQUENTIAL_IO [14.2.3], DIRECT_IO [14.2.5], and TEXT_IO [14.3.10] are implemented as described in the Language Reference Manual, as is the package IO_EXCEPTIONS [14.5], which specifies the exceptions that can be raised by the predefined input-output packages.

The package LOW_LEVEL_IO [14.6], which is concerned with low-level machine-dependent input-output, has not been implemented.

11 Characteristics of Numeric Types

11.1 Integer Types

The ranges of values for integer types declared in package STANDARD are as follows:

INTEGER	-32768 .. 32767	-- 2**15 - 1
LONG_INTEGER	-2147483648 .. 2147483647	-- 2**31 - 1

For the packages DIRECT_IO and TEXT_IO, the ranges of values for types COUNT and POSITIVE_COUNT are as follows:

COUNT	0 .. 2147483647	-- 2**31 - 1
POSITIVE_COUNT	1 .. 2147483647	-- 2**31 - 1

For the package TEXT_IO, the range of values for the type FIELD is as follows:

FIELD	0 .. 255	-- 2**8 - 1

11.2 Floating Point Type Attributes

FLOAT

DIGITS	6
MANTISSA	21
EMAX	84
EPSILON	2.0 ** -20
SMALL	2.0 ** 85
LARGE	2.0 ** 84 * (1.0 - 2.0 ** -21)
SAFE_EMAX	124
SAFE_SMALL	2.0 ** -125
SAFE_LARGE	2.0 ** 124 * (1.0 - 2.0 ** -21)
FIRST	-2.0 ** 124 * (1.0 - 2.0 ** -23)
LAST	2.0 ** 124 * (1.0 - 2.0 ** -23)
MACHINE_RADIX	2
MACHINE_MANTISSA	23
MACHINE_EMAX	127
MACHINE_EMIN	-128
MACHINE_ROUNDS	FALSE
MACHINE_OVERFLOWS	TRUE
SIZE	32

LONG_FLOAT

DIGITS	9
MANTISSA	31
EMAX	124
EPSILON	2.0 ** -30
SMALL	2.0 ** -125
LARGE	2.0 ** 124 * (1.0 - 2.0 ** -31)
SAFE_EMAX	124
SAFE_SMALL	2.0 ** -125
SAFE_LARGE	2.0 ** 124 * (1.0 - 2.0 ** 31)
FIRST	-2.0 ** 124 * (1.0 - 2.0 ** -55)
LAST	2.0 ** 124 * (1.0 - 2.0 ** -55)
MACHINE_RADIX	2
MACHINE_MANTISSA	55
MACHINE_EMAX	127
MACHINE_EMIN	-128
MACHINE_ROUNDS	FALSE
MACHINE_OVERFLOWS	FALSE
SIZE	64

11.3 Attributes of Type DURATION

DURATION'DELTA	2.0 ** -14
DURATION'SMALL	2.0 ** -14
DURATION'LARGE	131072.0 - 2.0 ** -14
DURATION'FIRST	-86400.0
DURATION'LAST	86400.0

12 Data Type Layout

12.1 Integer Types

INTEGER and LONG_INTEGER are allocated 16 bits and 32 bits respectively. For a user-defined integer type, the representation chosen is the smallest of the predefined integer types whose range includes the range of the declared type.

12.2 Enumeration Types other than BOOLEAN

A value of an enumeration type is represented by the integer value corresponding to its position in the type definition (positions being numbered from 0). The values of enumeration types are not signed. A maximum of 32768 elements can be declared in a type.

12.3 The Type CHARACTER

The type CHARACTER is represented as a user-defined type of less than 129 elements (integer range 0 .. 127, corresponding to ASCII.NUL .. ASCII.DEL).

12.4 The Type BOOLEAN

The values FALSE and TRUE are represented as the unsigned integer values 0 and 1 in a word.

12.5 Fixed Point Types

A value of a fixed point type is managed by the compiler as the value of *signed_mantissa * small*. *Signed_mantissa* is a signed integer. *Small* is the largest power of two that is not greater than the delta of the fixed accuracy definition. Thus fixed point types are mapped onto one of the two predefined integer types.

12.6 Floating Point Types

A single-precision floating point value is represented in the format:

bit 15 of first word:	sign of mantissa

the next 23 bits:	mantissa
bits 7 to 1 of second word:	exponent
bit 0 of second word:	sign of exponent

A double-precision floating point value is represented in the format:

bit 15 of first word:	sign of mantissa
the next 55 bits:	mantissa
bits 7 to 1 of fourth word:	exponent
bit 0 of fourth word:	sign of exponent

12.7 Array Types

Elements are allocated contiguously in memory. The size of an array is the product of the number of its elements and its element size.

12.8 Record Types

Record components are allocated contiguously in memory.

Implicit components are components created by the compiler within a record. They can be used to hold the record size, array or record descriptors or variant indices. The implicit component for the record size is in general laid out at the beginning of the record.

For dynamic components, a *dope* (pointer) is created where the component would normally have been laid out, and the component itself is allocated at the end of the record. The dope contains the offset from the start of the record to the actual component.

For a record type with variant parts, the common part an each variant part are laid out in the same way as described above, with the exception that dynamic components are laid out at the end of the entire record, not at the end of a specific variant part. Variant parts are overlaid when possible.

13 Other Implementation-Dependent Characteristics

13.1 Characteristics of the Heap

All objects created by allocators go into the heap. Also, portions of the Ada Run-Time Executive's representation of task objects, including the task stacks, are allocated in the heap.

All objects whose visibility is linked to a subprogram or block have their storage reclaimed at scope exit.

Use of UNCHECKED_DEALLOCATION on a task object may lead to unpredictable results.

13.2 Characteristics of Tasks

Each task, including the main program, is allocated two stacks. The primary stack is used for traditional subprogram linkage and for small local data. The auxiliary stack is used for dynamic-sized and large local data. The default primary stack size is the largest amount of free space in the data map after linkage (excluding MSEG) The default auxiliary stack size is the same as the primary stack size. These sizes may be controlled via Binder options.

Timeslicing is implemented for task scheduling. By using the Binder option SLICE, the time slice may be set to any period of 10 milliseconds or more. It is also possible to use this option to specify no timeslicing, i.e. tasks are scheduled only at explicit synchronisation points; this is the default. Timeslicing is started only upon activation of the first task in the program, so the SLICE option has no effect for sequential programs.

Normal priority rules are followed for preemption, where PRIORITY values run in the range 1 .. 10. All tasks with "undefined" priority (no pragma PRIORITY) are considered to have a priority of 0.

The minimum timeable delay is 10 milliseconds.

The maximum number of active tasks is limited only by memory usage. Tasks release their storage allocation as soon as they have terminated.

The acceptor of a rendezvous executes the accept body code in its own stack. A rendezvous with an empty accept body (e.g. for synchronisation) does not cause a context switch.

The main program waits for completion of all tasks dependent on library packages before terminating. Such tasks may select a terminate alternative only after completion of the main program.

Abnormal completion of an aborted task takes place immediately, except when the abnormal task is the caller of an entry that is engaged in a rendezvous. Any such task becomes abnormally completed as soon as the rendezvous is completed.

If a global deadlock situation arises because every task (including the main program) is waiting for another task, the program is aborted and the state of all tasks is displayed.

13.3 Definition of a Main Program

A main program must be a non-generic, parameterless, library procedure.

13.4 Ordering of Compilation Units

The Alsys HP 1000 Ada Compiler imposes no additional ordering constraints on compilations beyond those required by the language. However, if a generic unit is instantiated during a compilation, its body must be compiled prior to the completion of that compilation [10.3].

14 Limitations

14.1 Compiler Limitations

- The maximum identifier length is 255 characters.
- The maximum line length is 255 characters.
- The maximum number of unique identifiers per compilation unit is 1500.
- The maximum number of compilation units in a library is 1023.
- The maximum number of subunits per compilation unit is 100.
- The maximum size of the generated code for a single program unit (subprogram or task body) is 31 K-words (one code segment).
- There is no limit (apart from machine addressing range) on the size of a single array or record object.
- The maximum size of a single stack frame, including the data for inner package subunits which is "unnested" to the parent frame, can exceed 1 K-word; but in this case the user must override the 1 K-word limit default using a compiler option.
- The maximum amount of global data, including compiler-generated data, is restricted so as to fit into logical data space. The more global data there is, the less space is available for the primary stack. The "cut-off" size limit for global objects (above which they are allocated in VMA) can be controlled by compiler option. By default, it is 128 words.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are identified by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

NAME AND MEANING	VALUE
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	A....A1 |----| 254 characters
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	A....A2 |----| 254 characters
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	A....A3A....A |----| |----| 127 127 characters
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	A....A4A....A |----| |----| 127 127 characters
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that is is the size of the maximum line length.	0....0298 |----| 252 characters
$BIG_REAL_LIT A real literal that can be either of floating- or fixed-point type, has value of 690.0, and has enough leading zeroes to be the size of the maximum line length.	0....069.0E1 |----| 249 characters

NAME AND MEANING	TEST PARAMETER VALUE
$BLANKS A sequence of blanks twenty characters fewer than the size of the maximum line length.	235 blanks
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_647
$EXTENDED_ASCII_CHARS A string literal containing all the ASCII characters with printable graphics that are not in the basic 55 Ada character set.	"abcdefhgijklmnopqrstuvwxyz !$%?@[\]^'(}~"
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST	255
$FILE_NAME_WITH_BAD_CHARS An illegal external file name that either contains invalid characters or is too long if no invalid characters exist.	NO_BAD_CHAR_34567
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long if no wild card characters exists.	NO_WILD_CHAR_4567
$GREATER_THAN_DURATION A universal real value that lies between DURATION'BASE'LAST and DURATION'LAST if any, otherwise any value in in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST The universal real value that is greater than DURATION'BASE'LAST, if such a value exists.	10_000_000.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An illegal external file name.	TOO_LONG_NAME_111

NAME AND MEANING	VALUE
$ILLEGAL_EXTERNAL_FILE_NAME2 An illegal external file name that is different from $ILLEGAL_EXTERNAL_FILE_NAME1.	TOO_LONG_NAME_222
$INTEGER_FIRST The universal integer literal expression whose value is INTEGER'FIRST.	-32768
$INTEGER_LAST The universal integer literal expression whose value is INTEGER'LAST.	32767
$LESS_THAN_DURATION A universal real value that lies between DURATION'BASE'FIRST and DURATION'FIRST if any, otherwise any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST The universal real value that is less than DURATION'BASE'FIRST, if such a value exists.	-10_000_000.0
$MAX_DIGITS The universal integer literal whose value is the maximum digits supported for floating-point types.	9
$MAX_IN_LEN The universal integer literal whose value is the maximum input line length permitted by the implementation.	255
$MAX_INT The universal integer literal whose value is SYSTEM.MAX_INT.	2_147_483_647
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER if one exists, otherwise any undefined name.	NO_SUCH_TYPE

NAME AND MEANING	VALUE
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FF_FF_FF_FD#
$NON_ASCII_CHAR_TYPE An enumerated type definition for a character type whose literals are the identifier NON_NULL and all non_ASCII characters with printable graphics.	(NON_NULL)

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 19 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

. C32114A:	An unterminated string literal occurs at line 62.
. B33203C:	The reserved word "IS" is misspelled at line 45.
. C34018A:	The call of function G at line 114 is ambiguous in the presence of implicit conversions.
. C35904A:	The elaboration of subtype declarations SFX3 and SFX4 may raise NUMERIC_ERROR instead of CONSTRAINT_ERROR as expected in the test.
. B37401A:	The object declarations at lines 126 through 135 follow subprogram bodies declared in the same declarative part.
. C41404A:	The values of 'LAST and 'LENGTH are incorrect in the <u>if</u> statements from line 74 to the end of the test.
. B45116A:	ARRPRIBL 1 and ARRPRIBL 2 are initialized with a value of the wrong type--PRIBOOL_TYPE instead of ARRPRIBOOL_TYPE--at line 41.
. C48008A:	The assumption that evaluation of default initial values occurs when an exception is raised by an allocator is incorrect according to AI-00397.
. B49006A:	Object declarations at lines 41 and 50 are terminated incorrectly with colons, and <u>end case</u>; is missing from line 42.
. B4A010C:	The object declaration in line 18 follows a subprogram body of the same declarative part.
. B74101B:	The <u>begin</u> at line 9 causes a declarative part to be treated as a sequence of statements.

. C87B50A:	The call of "/=" at line 31 requires a use clause for package A.
. C92005A:	The "/=" for type PACK.BIG_INT at line 40 is not visible without a use clause for the package PACK.
. C940ACA:	The assumption that allocated task TT1 will run prior to the main program, and thus assign SPYNUMB the value checked for by the main program, is erroneous.
. CA3005A..D: (4 tests)	No valid elaboration order exists for these tests.
. BC3204C:	The body of BC3204C0 is missing.